# Integer Data Compression for Large Scale Computations

Mike Dvorak
Computational Mathematics Undergraduate
University of Minnesota - Duluth
Argonne National Laboratory
Argonne, Illinois 60439

September 28, 2000

Prepared in partial fulfillment of the requirements of the Office of Science under the direction of Dr. Mike Minkoff in the Mathematics and Computer Science Division at Argonne National Laboratory.

| | |
|---|---|
| Participant: | ______________________________ |
| | Signature |
| Research Advisor: | ______________________________ |
| | Signature |

**Abstract**

This project began with a need for the ability to compress a large data matrix of standard programming data types (integer and floating point types) into a smaller memory space in the hope of reduced computer file input and output (I/O). The project will deal with data compression three different ways: binary, numerical and encoded compression. Our initial work has been with the binary data type. Standard data types tend to use more memory than required to store the actual data. By removing most bits that do not contain data and compressing an array of data into a binary string, it is possible to reduce the amount of memory that is required to store an array. Some applications also do not require 100% accuracy. For these applications, it is possible to be within some certain error. This implementation includes the ability to specify absolute error as a compression parameter.

## Contents

## 1 Background and Introduction

Standard programming data types often are inefficiently used and can waste space in memory. Smaller programs can get away with this waste but large parallel programs need to optimize the way that they store their data in memory and on disk. The motivation behind this project is to find a way to run programs that have normally required an high amount of disk I/O to run either partially or exclusively in memory with a compressed data format. A natural place to start was with integer compression due to the homogeneity of the register storage space. Floating point types that conform to the IEEE Floating Point Standard [2] have separate parts for the sign, mantissa and exponent of the floating point number. Integers naturally only include the sign bit and the binary integer itself. Therefore it seemed like a natural choice to start compressing integer data on different machine architectures to gain confidence in our compression methods. Currently the integer compression will be useful to a large-scale quantum chemistry package named

COLUMBUS. [1]

Concurrently a sister project is being worked on at the DOE Ames Lab lead by Ricky Kendall dealing with floating point compression. Both research units at Ames Lab and Argonne National Lab share a common Concurrent Version System (CVS) code and documentation repository. This allows both teams to view each others work as well work on the same code directory in a secure manner. Both teams also communicate over the Multi-User Dungeon (the MUD) for collaborative purposes. This virtual reality text based program allows users to create virtual environment similar to that of the real world.

Our first testing with the binary string idea included compressing two integers of appropriate size into one 32-bit long integer (using ANSI C). This testing proved successful but did not allow much freedom of integer size or optimal compression. The next implementation includes a memory allocation of a integer array into which we planned to optimally compress the entire integer (lossless) into the array string. We ran into some challenging problems with this implementation with losing data while shifting bits over element boundaries.

Our current implementation included the ability for the user to compress the data in either a lossy or lossless fashion. The API that is explained below allows the user to specify an absolute error $\geq 0$. Earlier implementations wrote the compressed data out to hard disk on the local machine and then loaded the data back into memory. Later implementations may use memory mapping to cut back some of the associated disk I/O.

## 2 Application Programmer Interface

An important part of this project is the definition of the application programmer interface (API). As stated by Chen [3] in their API library specification, the compression library should have several design goals. These goals mainly stressed the importance of a portable implementation that allows the user to tailor the library to their needs at runtime (rather than at compile time). According to the API's project goals, the user should not have to worry about where the compressed data is stored (memory or permanent storage).

[1]http://www.itc.univie.ac.at/ hans/Columbus/columbus.html

## 2.1 Integer Compression Library Overview

The API has the ability to compress both floating-point and integer data matrices. Initial implementations will only include the ability to compress single dimensional arrays leaving the user to deal with the implementation details of a multidimensional array. Since this paper deals mainly with integer data compression, the API for the compression and decompression function are shown below:

```
int compress(int origArr[], const int numElem,
BIT_FILE *BitOutputFile, const int absError, int *compByteLength);

int decompress(BIT_FILE *InputBitFile, int decompArr[],
const int numElem, const int absError, int const *compByteLength);
```

## 2.2 Integer *compress* Function

The *compress* function takes an integer array that is in memory (origArr) of size numElem and compresses it out to a BIT_FILE[2] pointer *BitOutputFile in the local directory with absolute error less than or equal to the integer absError.[3] The pointer to the compByteLength integer tells the compression algorithm the size of the numerical words in the file. The user can specify zero for the absolute error if they wish to obtain lossless compression.

## 2.3 Integer *decompress* Function

The *decompress* function uses the same parameter specifications that the compress function requires. The array undergoes the reverse of the compression process. Most importantly, the bits undergo a *left* shift when being read back into the memory array. This is where the acceptable absolute error is introduced into the elements.

[2]as defined by Nelson [4]

[3]Absolute error is defined as $\Delta x \equiv x_o - x$

# 3 Integer Compression Implementation

## 3.1 ANSI Compiler Definition of Integer

The ANSI C Standard [1] specifies several different parameters for compliant compilers. Adherence to this standard is very important to this project since this API will be running on several different architectures. Among the most important aspects are the dimensions of various integer types. Several constants are included in the $< limits.h >$ file which is included with each C compiler. The ANSI C Standard [1] states "[The type's] defined values shall be equal or great in magnitude (absolute value) to those shown, with the same sign." The table below list some of these relevant values:

| **Constant** | **Value** |
|---|---|
| INT_MIN | -31767 |
| INT_MAX | +32767 |
| UINT_MAX | 65535 |
| LONG_MIN | -2147483647 |
| LONG_MAX | +2147483647 |
| ULONG_MAX | 4294967295 |

These values were important considerations for the implementation because the integer values must be checked to ensure that data is not corrupted by underflow and overflow conditions.

## 3.2 C Implementation

Most operations performed in C are bytewise meaning that they are oriented to eight bits. Most work in compression is performed bitwise. This leads to some special challenges that must be overcome when trying to compress and decompress data in memory and on disk. Fortunately, other work has been done in this area. Nelson [4] provides a very nice implementation of a *bitio* library in C. This implementation sets up a data structure for bit-files:

```
typedef struct bit_file {
    FILE *file;
    unsigned char mask;
    int rack;
    int pacifier_counter;
} BIT_FILE;
```

Besides the obvious open/close functions, four main functions *(void InputBit, void InputBits, int OutputBit, int OutputBits)* are provided for file I/O. These functions allow the user to read/write to storage without worrying about some of the bitwise specifics. While the *bitio* library provided us with a basic bit-file operations, other bit operations were still necessary. The standard bit-shift and bit-arithmetic operators were used extensively (with care though) through out the implementation.

During testing, it was necessary to create large arrays of random integers, calculate the number of bits that were required for a integer and do a few other miscellaneous routines. A *intCompressAbs* library was created for these tasks. Included in the file was also the current implementation of the *compress* and *decompress* functions which varies in a few parameter values from the API at this time.

## 3.3 Compression Algorithm

An accurate and *lightweight* algorithm is crucial for the performance of the compression software. Although many lossy and lossless data compression algorithms exist i.e. Huffman, Adaptive Huffman, Arithmetic, LZ77, etc..., many of these programs do not take advantage of the C integer and floating point types. For example, some applications i.e. computational chemistry problems, use sparse matrices that have many zero entries. A large scale sparse matrix may included for example one million consecutive zero entries. It does not make sense to encode all of these entries individually. Instead it makes much more sense to include in the compressed data a statement that say "the next million entries are zeros". By not using one of these standard compression algorithms, you are given the ability to add custom functionality to the data compression.

The compression algorithm that was used for the integer compression that is talked about in this paper is listed below:

1. Find the maximum value in the array that you are going to compress.

2. Given a specific acceptable absolute error, find out how many bits are required to represent that absolute error.

3. Read in an integer that is to be compressed.

4. Bit-shift the integer to the right the required number of bits and write out to file or memory only the $|maxInt| - |absError|$ for that integer.

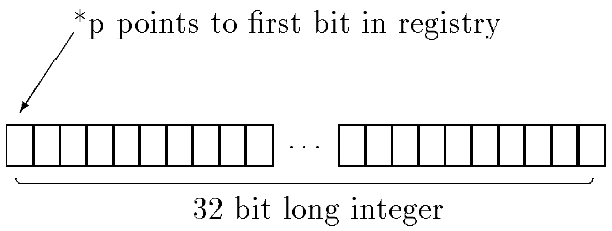

Figure 1: Standard piece of integer memory.

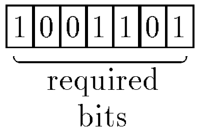

Figure 2: Integer using only a minimal amount of space in memory.

5. Repeat step 1-6 until the entire array has been compressed.

Step 2 is necessary for lossy compression of integers. The user specifies an acceptable amount of error in the *compress* function call. The number of bits required to represent the data is then calculated. For example, if the amount of acceptable error was $\pm 8$, then three bits ($2^3$) could be lost during compression. Figure 1 shows a standard piece of integer memory most programming environments. Figure 2 shows a piece of memory that uses only the minimal amount of space. This integer can be further reduced by shifting it the acceptable number of error bits to the right.The only thing that needs to be written out to disk or memory is the first bits up to the point where the absolute error is relevant.

The end result of compressing the integer array is that a large binary string is created either in memory or storage that contains the compressed array. Figure 3 shows a picture of several integers going into this compressed string. This binary string can be decompressed with the *decompress* func-

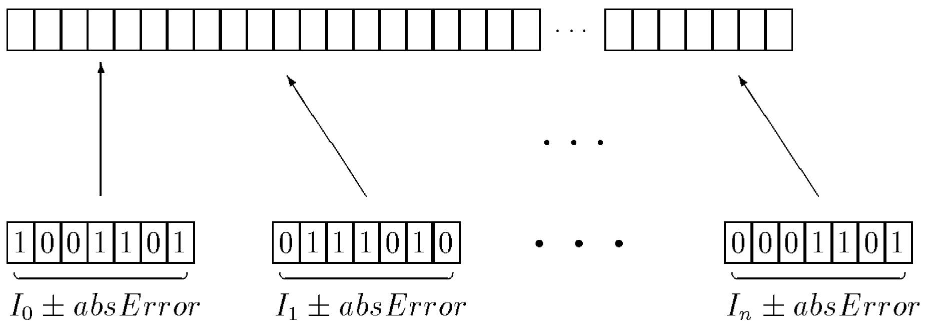

Figure 3: Binary string of compressed integer data.

tion. Ideally a header would be included in the file containing the compression parameters i.e. word length and the amount of absolute error to allow for automatic decompression. Future implementations may provide these features.

## 3.4 Quality Assurance

Throughout this project, it was necessary at times to ensure that the compression and decompression was being done within the specifications of the project and the API. Small testing programs were made that compared the original and compressed data. As the project develops, more rigorous testing methods will be devised.

## 3.5 Compression Performance Analysis

After completion of the absolute error module, the program was executed on three different machines at the High Performace Computing Research Facility (MCS) at Argonne National Laboratory (ANL). The three differenct machines that were used for testing at ANL were a Intel Celeron Linux workstation, a 28 node Origin 2000 / Reality Monster (Denali), and a 512 CPU Linux cluster (Chiba City). Although this program is run sequentially, the three different machines provided a way to compare results on different machine architectures. The testing consisted of using a Perl script to run through several different parameter values and array sizes. The table below gives a sample of some of the data that was obtained from *Chiba City*:

| **Parameter** | | | **File size (kb)** | | **Time (sec)** | |
|---|---|---|---|---|---|---|
| testSize | maxIntSize | absError | uncmp | cmp | reg read | cmp read |
| 240000 | 30000 | 10 | 960 | 360 | 0.527466 | 0.264599 |
| 240000 | 30000 | 100 | 960 | 270 | 0.515900 | 0.204616 |
| 240000 | 30000 | 1000 | 960 | 180 | 0.515882 | 0.146407 |
| 240000 | 30000 | 2000 | 960 | 150 | 0.508485 | 0.126291 |
| 240000 | 30000 | 4000 | 960 | 120 | 0.527156 | 0.105392 |

These results were somewhat interesting. By accepting an error of $\pm 1000$ ($\pm 3.33\%$) it was possible to reduce the array read time from file by more than a factor of 3. File size was also reduced by more than a factor of 5. Due to the very similar natures of the *compress* and *decompress* functions, run times for these functions were virtually identical in testing.

## 4 Continuing Research

The use of encoding is the last way that we intend to compress matrix data. By identifying elements in the matrix that appear more often than others, it is possible to give these common elements an abbreviated notation. Common algorithms exist to do such compression i.e. the Huffman Algorithm. Using a variant of this algorithm, it will be possible to compress the binary string into yet a smaller piece of memory.

The remainder of this research project will focus on the completion and testing of the integer compression library. An existing chemistry application will provided some benchmark times to compare the performance of the existing data storage methods verses the newer compressed methods.

Future implementations will include both the ability to compress both floating point and integer arrays. The final API will allow any programmer the ability to compress and decompress data seamlessly and optimally in their applications.

## 5 Acknowledgements

I would like to thank my advisor Dr. Mike Minkoff of the Mathematics and Computer Science Division (MCS) at Argonne National Laboratory for his guidance and support. I would also like to thank Ricky Kendall at Ames Laboratory and Ron Shepard at ANL for the joint project support. Thanks also goes to the Department of Energy, the National Science Foundation, MCS at Argonne and Argonne National Laboratory for creating, organizing and funding the program. Finally I would like to thank the High Performance Computing Research Facility at Argonne for the use of their exceptional computing facilities.

## References

[1] American National Standards Institute. *American National Standard for Information Systems: Programming Language - C*, 1989.

[2] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Binary Floating Point Arithmetic.* 345 East 47th Street, New York, NY 10017, USA, MAR 1985. ANSI/IEEE Std 754-1985.

[3] Kendall, R., Shepard, R., Minkoff, M., Dvorak, M., and Chen, W. A generic compression library for high performance distributed applications.

[4] Nelson, M., and Gailly, J.-L. *The Data Compression Book.* M&T Books, 1996.